Mini-Project 1

Computer System Performance
KSCOSYP1KU

Wednesday 17th May, 2023

Claudio Sotillos Peceroso
clap@itu.dk

Ívar Óli Sigursson
ivas@itu.dk

Jorge del Pozo Lérida
jord@itu.dk

I. Introduction

Data partitioning on Chip Multiprocessors refers to the process of dividing data into smaller pieces, and distributing those pieces to multiple processors in a parallel processing system. This can be done for a variety of reasons, such as to improve query performance, enable parallel processing of the data, or reduce the amount of data that needs to be processed at once.

There are four common methods of data partitioning on Chip Multiprocessors:

1) Independent Output: In this method, each processor works on its own part of the data and generates its own output. This method is useful when the processors do not need to communicate with each other during the processing.
2) Concurrent Output: This method is similar to Independent Output, but the processors communicate with each other and share their results in real-time. This method is useful when the processors need to work together to produce the final result.
3) Count-Then-Move: In this method, one processor is responsible for counting the number of elements in the data, and then distributing those elements to the other processors. This method is useful when the processors need to process the data in a specific order.
4) Parallel Buffers: In this method, the data is divided into smaller pieces, and each processor is assigned a buffer to store its portion of the data. This method is useful when the processors need to access the data randomly.

In this research project, we have chosen 2 of the partitioning methods listed above. The data partitioning techniques chosen are **Independent Output** and **Concurrent Output**.

Besides, in order to assign each data instance into a certain partition, methods such as Range Partitioning, Hash Partitioning [1] and Round-Robin Partitioning are used.

We use a hash partitioning technique called multiplicative hashing. This technique consists of generating a unique hash value for a given entry. In the context of data set partitioning, multiplicative hashing is often used to assign data points to specific partitions based on their hash values. It is implemented as follows. Firstly, select a prime number 'p' that is greater than the maximum number of partitions required. Secondly, choose a multiplier 'a' that is less than 'p'. Thirdly, for each data point, compute its hash value using the formula $h(x) = ((a*x) \mod p) \mod n$, where x represents the data point, 'n' is the number of partitions and mod is the modulo operator. Finally, assign each data point to the partition that corresponds to its hash value.

For example, let's say we want to partition a data set of 100 data points into 4 partitions using multiplicative hashing. We can choose p = 101 (a prime number larger than 4) and a = 7 (less than 101). Then, for each data point x, we can compute its hash value using the formula:
$h(x) = ((7 * x) \mod 101) \mod 4$
The result will be a value between 0 and 3, which corresponds to the partition number that the data point should be assigned to.

II. Experimental methodology

We will now proceed to describe our experimental setup following the steps of the experimental design that we discussed in our first lecture.

- Goal: Let's start by defining the goal of our project, which is to make measurements to compare two out of

[1] **Hash P:** Involves partitioning the data based on a hash function that assigns each record to a specific partition based on its hash value.

the four data partitioning techniques previously stated, concretely we will compare the *Independent Output Technique* **VS** the *Concurrent Output Technique*.

- System: From the hardware point of view we will use the server already described (we will not compare two different systems), and from the software point of view we will use the 2 data partitions already mentioned.

- Metric: In order to compare the two data partitioning techniques, we have considered it more appropriate to use tuples generated per second as the main metric. For the time measurements, we have used the c++ *"chrono"* library.

In addition, using the **perf command** (the linux performance analysis tool), we have measured Context Switches [2], Cache Misses, dTLB load Misses, and iTLB load Misses [3].

- Parameters (k): The parameters that we will take into account in the experiments will be the hash bits, the threads, and the page size.

- Levels (l): With respect to the hash bits, these will take values from 1 to 18, representing the number of partitions. Regarding the number of threads, these will take the following values (1, 2, 4, 8, 16, 32). Finally, the page size will only take two possible values (4 Kb or 2 Mb), as these are the ones supported by our server (there are other servers that support more values).

- Type of Experiment: Mainly measurements will be used to evaluate the experiments, plus a slight analytical touch due to the use of the measurements provided by the "perf events".

- Workload: Since we are working with our own created data, we are dealing with a Synthetic Workload, using it to benchmark our System under two concrete Scenarios.

- Experimental Runs: We considered it optimal to carry out 8 experiments for each partitioning technique, and then average the results of these experiments to obtain more accurate results.

Each experiment consists of measuring the number of tuples per second that the server is able to generate given a number of threads and a number of hash bits. In other words, an experiment is not finished until all combinations of threads and hash bits have been completed, measuring their respective performance.

EXPERIMENTAL SETUP SUMMARY

- **Goal:** *Independent Output Technique* **VS** the *Concurrent Output Technique*
- **System:** Linux dasya1
- **Metric:** Million Tuples per Second, Context Switches and Cache Misses
- **Parameters(k):** Nº Hash Bits , Nº Threads and Page Size
- **Levels(l):** Hash B: 1,...,18; Threads: (1,2,4,8,16,32); and Page Size: 4 Kb or 2 Mb
- **Type of experiment:** Measurements mainly
- **Workloads:** Synthetic Workload
- **Experimental Runs:** 8 Exp Runs

All experiments were conducted on the University server (Linux dasya1 4.15.0-200-generic 211-Ubuntu SMP). It has 32 CPUs (CPU Type: AMD Opteron(tm) Processor 6386 SE, 1400 MHz) arranged in 4 NUMA nodes. Each CPU is able to handle two threads.

- NUMA node0 CPU(s): 0-7
- NUMA node1 CPU(s): 8-15
- NUMA node2 CPU(s): 16-23
- NUMA node3 CPU(s): 24-31

Other relevant information regarding the server and the implementation details is shown in the following table.

Clock rate	1381.098 MHz
Cores (Threads/core)	32 (2)
RAM	126 GB
L1 Data Cache	16K
L1 Instruction Cache	64K
Shared L2 Cache	2048K
L3 cache:	6144K
Page Size	4 KB or 2 MB
Operating System	Ubuntu 18.04.6 LTS
Programming Language	C ++
Hashing Technique	Multiplicative Hashing

TABLE I: Specifications of our system

The input data in both approaches is an array of length 2,000,000 consisting of 64-bit unsigned integers (u64). Both approaches use multiple threads to process the large input data array. Each thread is assigned a specific range of input data to work on.

To carry out all the experiments, we created a shell script called *run.sh*. In it, we iterated over all com-

[2]**Context Switches :** This occurs when the operating system changes the current execution context of a process or thread in order to allocate system resources to another process or thread. The current state of the process or thread is saved in memory and can be restored later.

[3]**Cache Misses/ dTLB load Mises/ iTLB load Misses :** These three cache misses occur because a processor needs to fetch a piece of data/ a mapping between virtual and physical addresses/ an instruction that is not currently stored in the cache memory/ dTLB cache/ iTLB cache, respectively. In these cases, the processor must access the slower main memory to retrieve the required information, which can result in longer latency and lower performance.

binations of parameter levels (for both approaches) as input to our executable, which was a compiled file derived from our source code in *C++*. We used Linux command-line tool *perf* at each iteration to record cache-misses, dTLB-load-misses, iTLB-load-misses and context-switches of every run. Finally, we created a *Python* script to parse output files and plot all our results.

All code and results obtained throughout the project can be found in ComputerSystemsPerformance.

III. Interpretation of Results

Results for different metrics obtained for both methods can be seen in Figure 1, where the value of such metrics is plotted for combinations of different number of hash bits and number of threads used. Discussion about each method will be done in sections III-A and III-B, while comparison between them in section III-C.

A. Independent Output

Initially, what we did was; create the partitions, decide the range of values that each thread would deal with, and initialize each thread with this information. Subsequently, we measured the performance exclusively on the lines of code responsible for "thread joining". However, we realized that this way of measuring performance was not accurate, as by the time we started the timer some threads would have already started to perform calculations resulting in shorter time measurements.

To address this issue, we modified our approach, saving all the information that each thread needed to perform its task (start end indices of the range of values the thread is in charge of, hash bits in this particular iteration, and its respective output buffer).

With this modification, we can initialize all threads simultaneously, measuring only the computation time used by the threads to compute the output partition to which each tuple belongs.

We can see the results of the definitive approach in the L1 graph of Figure 1. Independent output achieves very high performance for low values of hash bits, but as the hash bits increase there is more metadata overhead and performance decreases. Between 1 and 10 has bits the number of millions of tuples per minute decays exponentially. It is in this first half of the X-axis where a clear improvement can be seen as the number of threads increases (in particular, the improvement is noticeable from 4 threads upwards). From hash bit 10 onwards, there is not much difference in performance as the number of threads increases (there is a slower decay in throughput).

Furthermore, if we look at the four graphs (L2-L5) below this one (which show the metrics measured with *perf* for the Independent Output approach), they help us understand why there is a noticeable change in the number of millions of tuples per second from around hash bit 6. If we look closely, it is not until hash bit 6 that the curves start to detach from values close to 0 (this can be seen more clearly in the first 3 graphs, the last one only shows notorious changes in the last hash bits).

An increase in any of these four metrics means that execution will slow down, either because the processor has to spend more time searching for certain information in main memory (increase in the number of cache misses, any of the three kinds), or because thread states are being saved and restored more frequently (increase in context switches).

B. Concurrent Output

If we look at the R1 graph of Figure 1 we observe that, as expected, Concurrent output performed worse for a low number of hash bits. This is due to the fact that this approach requires inter-thread coordination, which can be counterproductive when the number of threads used to is higher than the number of partitions, requiring excessive coordination between threads. This can be observed in the R2 graph, whereas the number of hash bits increases, the amount of context switches decreases exponentially since threads are more spread across the partitions and do not conflict so much. We see also that this amount remains almost stable as hash bits increase.

Since this approach requires thread coordination, the metric that best explains the performance behavior is the Context Switches shown in the R2 graph. If you look at this graph, it is somewhat inverse to graph R1. Between hash bits 1 and 6 we can see an exponential drop implying that for those hash bits the performance is much worse. Between hash bits 6 and 12 we can see a valley which is reflected in a sort of Gaussian in the R1 graph. Then, between hash bits 12 and 18 the number of context switches increases again causing a valley in the R1 graph. From hash bit 18 onwards, the number of context switches goes down again, but there is no noticeable increase in performance in the R1 graph (except for the execution with 32 threads, which does improve its performance). We believe that this is due to the high number of dTLB misses (graph R4) that occur from hash bit 18 onwards, which, although it is true that it may not have as much impact on performance

as context switches may have in this scenario, it has to affect something when reaching such high numbers.

C. Independent vs. Concurrent Output

For a low number of hash bits, Independent output outperforms Concurrent output with only a few threads used, since Concurrent output stays behind due to a large number of context switches. But then we see that between 8 and 12 has bits, Concurrent output scales better reaching a peak of performance around 9 has bits.

However, for both approaches throughput decreases at some point inevitably, as the number of partitions increases. By looking at dTLB Load misses in the fourth row of Figure 1 (L4 and R4), we see that both approaches suffer from this phenomenon, decreasing their throughput for a large number of hash bits. Due to the large amount of virtual addresses needed to handle such a big number of partitions, a full translation process is needed to retrieve the physical addresses from the main memory, thus decreasing performance.

IV. Concluding remarks

After carrying out this work we have realized the large number of variables that can affect the performance of a given program, having a clearer picture of the processes that processors carry out at a low level without us realizing it and that can slow down our system quite a lot. Depending on the task we want to carry out, we must be very careful when parallelizing it in order to speed it up, as a bad choice of levels can cause an execution with a performance that is far from the optimal performance that we could obtain.

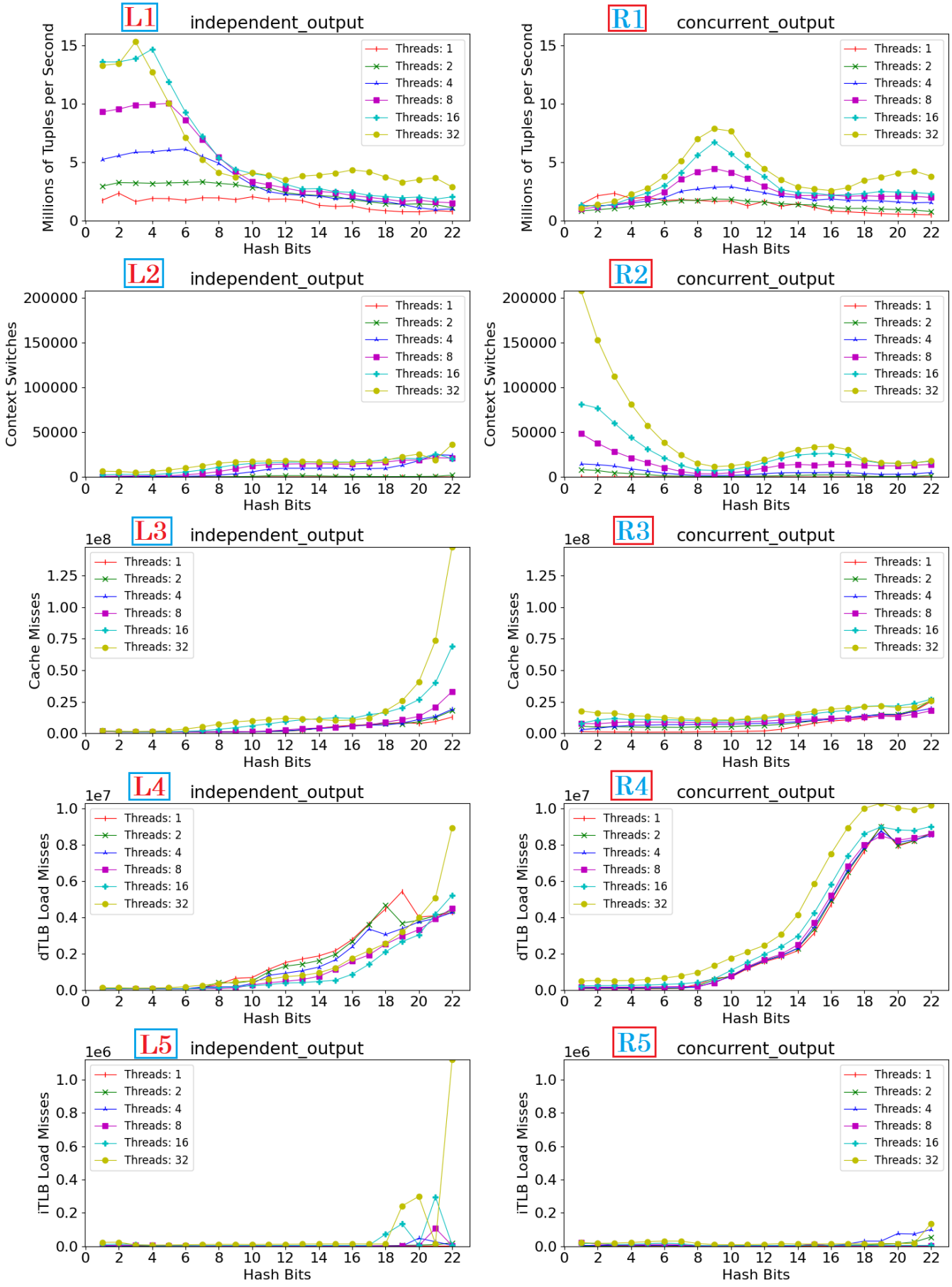

Fig. 1: Metrics obtained for both Independent output (left) and Concurrent output (right) approaches. All results are the average of 8 runs. In the upper left corner of each graph there is an identifier that we will use to refer more easily to these (L $\rightarrow Left; R \rightarrow Right$).